AD-A250 208

(2)

ENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	31 May 92	Final, 01 Nov 90 - 31 OCT 91

4. TITLE AND SUBTITLE
Development of a High-Performance Fortran Compiler for Porting CFD Codes to the Navier-Stokes Computer

5. FUNDING NUMBERS
AFOSR-91-0003
2307/A5

6. AUTHOR(S)
Daniel Rosenchuck

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Princeton University
PRINCETON, NJ 08544

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 92 0389

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
DIRECTORATE OF AEROSPACE SCIENCES
BOLLING AFB, DC 20332-6448
NA

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFOSR-91-0003

11. SUPPLEMENTARY NOTES

DTIC
ELECTE
MAY 19 1992
S A D

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Research on developing a high-performance FORTRAN compiler for the Navier-Stokes Compute (NSC) was performed during the contractual period. The thrust of the work was to develop a prototype compiler the NSC MiniNode. The NSC MiniNode is an operational prototype hardware node, that represents the key building block of a parallel-processing supercomputer whose architecture was designed to support the efficient simulation of large-scale complex fluid flows.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Fortran, Compiler, Navier-Stokes, CFD	7
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

Research on developing a high-performance FORTRAN compiler for the Navier-Stokes Compute (NSC) was performed during the contractual period. The thrust of the work was to develop a prototype compiler the NSC MiniNode. The NSC MiniNode is an operational prototype hardware node, that represents the key building block of a parallel-processing supercomputer whose architecture was designed to support the efficient simulation of large-scale complex fluid flows. The NSC is based on a small number of powerful nodes, where each node *running standalone* is comparable in sustained performance to current supercomputers. Each node has dynamically reconfigurable internal systolic arrays (arithmetic logic structures, or ALSs) connected via crossbar switch to multiple independent memory and address-generation modules (memory planes). Three kinds of ALSs can be used in the node: singlets with one floating-point unit, doublets with two floating-point units, and triplets with three. These computational and storage assets support the multiple levels of parallelism required for efficient solution of most numerical forms of the Navier Stokes Equations. The ALSs alone provide fine-grained support, while the node as a whole represents medium-grain hardware parallelism. Multiple nodes that are interconnected provide coarse-grain support for global domain decomposition. Overall, the architecture may be described as a continually reconfigurable Very Long Instruction Word (VLIW) machine.

The number of memory and ALS assets are used to categorize NSC nodes of various sizes. The convention is to specify the number of singlets, doublets, and triplets and memory planes in a node. Thus, an x:y:z/m configuration has x singlets, y doublets, and z triplets with m memory planes. At present, the prototype MiniNode exists in hardware at Princeton University, and has a 0:2:2/4 configuration (i.e. two doublets, two triplets — 10 total floating-point units — and 4 memory planes).

A generic prototype compiler to efficiently port code to the NSC was studied for suitability in specifically compiling FORTRAN code for the NSC MiniNode. The compiler accepts unmodified 'dusty deck' ANSI FORTRAN 77 source code and optimizes variable storage in memory to minimize reference conflicts, and maximizes the average number of floating-point and integer/logic processors utilized over the course of a flow simulation. This compiler has several novel features in the general area of compiler design. These features include:

1. direct creation of dependency graphs from the unmodified source code,

2. high-level approximate modelling of various elements of the target computer architecture (in this case, the NSC), a short list of which includes:

   (a) memory address computation unit,

   (b) memory plane and cache architecture, size and update/replacement algorithms,

   (c) ALS external and internal data paths, registers and execution modelling,

3. perfo ance prediction of candidate code fragments produced by the compiler using the models from Item 1 above to provide feedback to the compiler during optimization, and

4. heuristics aimed at recognizing and efficiently implementing computational constructs frequently encountered in CFD algorithms (such as linear matrix operations, FFTs, conditional evaluations for numerical stability, etc.).

The compiler is parameterized, and is dubbed a Parameterized Memory/Processor (PMP) optimizing compiler. The basic features of a parallel computer architecture, such as the number, type, and behavior of memory, processors, registers, control stores, and their interconnections and couplings, are parameterized. This permits the study of the suitability of existing and proposed parallel computers in handling large scientific codes. It also provides a means by which architectural variations may improve performance, such as the addition of additional ports to memory for example.

It was determined that the compiler is indeed suitable for the NSC MiniNode, and that it may be used, with further development, to port large 'production' CFD codes to the full multinode NSC. This is the subject of ongoing AFOSR-supported research, and is expected to culminate with hardware and software CFD-code demonstrations on a multinode NSC.

Several of the key research results are presented in the attached preprint which will appear in the Proceedings of the 1992 Scalable High Performance Computing Conference, IEEE Computer Society Press.

Parameterized Memory/Processor Optimizing FORTRAN Compiler for Parallel Computers

Daniel M. Nosenchuck
Department of Mechanical and Aerospace Engineering
Princeton University
Princeton, New Jersey 08544

Abstract

A new approach to generating low-conflict, parallel instructions for complex applications is introduced in this paper. This method is presented within the context of a FORTRAN compiler. An approximate simulator has been incorporated within a parallel-code/ domain-decomposition loop within the compiler. The simulator estimates the performance of candidate instruction segments, and guides the selection of appropriate code transformations, heuristics, and data storage strategies. At present, many aspects of the target machine are parameterized, to permit investigations of a number of parallel-computer architectures. In this paper, the compiler is illustrated for a Navier-Stokes Computer target node application.

1 Introduction

The generation of efficient code coupled with automatic data decomposition remains one of the key unsolved problems in parallel computing. Large-scale scientific simulations require partitioning of massive data arrays, often with complex structure, among multiple memory units when run on a parallel machine. For rapid throughput, parallel execution must dominate sequential processing. This requires code sequences and data storage to provide nearly conflict-free, continuous, simultaneous data access. To date, this need is generally met by distributing data via explicit directives in source code, with subsequent compiler-generated parallel code. 'Optimization' is often by hand, in a laborious off-line process driven by profiler output.

The present work was motivated in part by the need to port efficiently large scientific simulation programs[1] to the Princeton Navier-Stokes Computer (NSC)[3]. So-called 'production' codes are often written in FORTRAN-77, or earlier versions with little vectorization and no parallel construction. The goal of the compiler is to generate machine instructions from standard (unembellished with embedded directives) FORTRAN-77 source, that would execute at a minimum of 50% of the peak NSC speed.

The compiler automates and optimizes domain decomposition and parallel code generation, is general in nature (i.e. not restricted to the NSC architecture), and is dubbed a parameterized memory/processor (PMP) optimizing FORTRAN Compiler, or PMPC. A specific implementation on the NSC has been made.

This new compiler architecture followed largely in part from years of experience hand-coding applications on the NSC using low-level tools. Initial development followed from attempted automation of both the routine and the creative aspects of manual, low-level programming.

2 Overview of Novel PMPC Elements

2.1 Compile-time Simulation

While most scientific applications are inherently rich in parallelism, a careful balance must be struck between the formation of parallel code and the formation and distribution of data structures. This ensures that subsequent data access does not result in a large number of conflicts, decreasing the performance of the target machine. Since many potential conflicts cannot be identified *a priori*, the central feature of the PMPC is an iterative, optimizing procedure. Candidate code segments are formed and identified through the use of basic code transformations and extensive heuristics. These code-segments typically relate to scalar sections, loops, subroutines, etc., and are generally 100 source lines or less in length. The code block is then tested for execution efficiency by an *approximate simulator* module, which invokes an approximate architectural model of the target computer. The simulator is designed to run rapidly at compile time, and provides a rough simulation of only those portions of the computer that may lead to data-flow bottlenecks, such as the memory, cache, and processor units.

The simulator *estimates* performance by only simulating that portion of code which may result in ambiguous memory references or subsequent conditional tests. The domain of the simulation is constrained significantly to maintain adequate overall compiler throughput. Thus, the input and initialization vectors and arrays[2] represent only small subsets[3] of com-

[1] An initial area of application of the compiler is computational fluid dynamics (CFD), with codes typically consisting of $10^3 - 10^5$ source lines.

[2] These data structures often represent the initial and boundary conditions of a scientific simulation.

[3] Typically less than 100 elements of a data structure which

plete run-time arrays. One particularly interesting result, discussed below in NSC Applications, is that the statistics of the simulation were not strongly dependent on the particular makeup of the initialization arrays. The purpose of the simulator is to provide some guidance to the optimizer for the selection of appropriate code transformations and heuristics. The simulator is not meant to give an accurate assessment of performance.

If the estimated performance is less than a predetermined level, the compiler is guided by the simulator output-statistics in selecting a new code transformation, or modifying the data storage strategy. For example, when data-access conflicts are uncovered (often due to ambiguous data references), the simulator outputs memory reference-, conflict- and primitive-operation-counts. The derived performance estimate determines whether optimization should continue on the same code fragment, or if the next code segment should be generated and analyzed. If the performance is inadequate due to insufficient processor utilization or data conflicts, data structures can be moved or redistributed (based on the access patterns and conflicts uncovered by the simulator), and/or the code fragment can be modified. Standard code transformations along with heuristics such as loop-splitting or fusion, are incorporated in conjunction with data movements.

2.2 Wide Intermediate Code Format

The format of most scientific and other application source-codes suggests the parallel and/or systolic[4] execution of operations. In preserving the high-level format, formation of parallel or systolic code constructs from a Wide-format Intermediate Code (dubbed WIC) representation is more natural than having to reconstruct such constructs from sequential code (see below). The high-level language (FORTRAN) is translated after parsing and dependency analysis into an internal WIC. In essence, the format of the WIC inherently maintains local parallel and systolic constructs and dependencies found in the original source code. The natural relationships between operand fetches, complex intermediate operations, and result storage are preserved within the WIC statements. A single line of WIC code often directly follows from discrete source-code lines. The burden on subsequent analysis to extract possible parallel or systolic implementations is lessened. WIC may be contrasted to ubiquitous sequential internal code-formats, typically characterized by simple load, move, operate, store sequences. This latter format places an increased burden on the parallel code analyzer which must reconstruct many of the 'obvious' parallel code elements that were explicit in the original source code.

The WIC code embodies all of the actions directed by the source program. In addition, it maintains symbol-table attributes and local data dependencies.

An example of the basic WIC-statement format is

$$\text{Result} = (\textit{Oper}\ 1 \odot_1 \textit{Oper}\ 2) \odot_2 (\textit{Oper}\ 3 \odot_3 \textit{Oper}\ 4)$$

where $\odot$ signifies an arbitrary high-level operation, such as $+, -, \times, \div$, and *Oper* signifies a operand, either from memory, a register, or from the result of a preceding computation. In this example, the WIC shows the local dependencies (based on parenthetical ordering), where $\odot_1$ and $\odot_3$ may execute in parallel, with subsequent processing by $\odot_2$.

To best illustrate an example of one particular embodiment of the format of the intermediate code, consider a simple systolic vector operation, as extracted from a test kernel:

```
      do 11 1=1,imax
         do 10 j=1,jmax
            z(j)=const1*b(j)-(const2*a(i)+
                              const3*c(i,j))
10       continue
11    continue
```

The internal WIC generated for the above code is shown below:

```
p00:=$8#11RS*$1#5MA1+$9#11RS*$4#8MA1 %(1)
==$13#12MA1==$7#11RS*$2#6MA1-p00      %(2)
^>                                    %(3)
```

where line (1) evaluates the parenthetical expression, line (2) completes the evaluation of `z(j)` and the line (3) symbol indicates END DO. WIC uses symbol-table mnemonics particular to the present embodiment of the compiler for the NSC. The potential for multiple independent memory planes is reflected in the structure of the token. The format of the data-structure symbol-table tokens is described in Table 1.

Data-structure symbol-table tokens:
`$ ! xx #mp stor occ`, where:

Symbol	*Explanation*
$	data element
!	Scatter/multistore indicator
xx	variable reference number
#mp	memory plane number
stor	disposition of variable given by
	MA: memory-based array
	MS: memory-based scalar
	RA: register-based delayed array
	RS: register-based scalar
occ	variable occurrence number

Table 1: Format of Data-structure Symbol Tokens

As shown in the example above, WIC is essentially comprised of nested interior dependency nodes (within the DO-loop). (The loop header code was eliminated for simplicity.) Here, non-terminal internal node p00 indicates a *systolic phrase*. The phrase-break was merely driven by the parenthetical ordering indicated in the source. The token bounded by == represents the root of the local dependency tree. Thus, as illustrated by this simple example, the WIC presents a

could range up to 10^8 or more elements.

[4] Systolic execution is operationally defined by considering data streams that enter an array of processing elements whose outputs are directly fed into subsequent processor inputs. Data is thus processed in an assembly line fashion, without the need for intermediate storage.

natural ordering of intermediate and final results that lend themselves to relatively straight-forward subsequent analysis and parallel implementation.

Clearly, the inherited attributes can be parsed much finer where, in the limit, conventional sequential intermediate code, as discussed above, would result. However, this would require increased work from the code analyzer, and might result in lower parallel performance relative to that expected by WIC analysis. It should be noted that WIC ordering does not significantly constrain subsequent systolic and parallel code generation and optimization.

3 PMPC Implementation on the NSC

For the NSC application, a Node architecture (see [1] for example) was specified. Main parameters were memory size and configurations, and internal processor count. The NSC convention is to specify the number of so-called singlets, doublets, triplets [5], and memory planes in a node. Thus, an x:y:z/m configuration has x singlets, y doublets, and z triplets with m memory planes[6].

The simulator module was a small, streamlined subset of a complete single-node NSC simulator [2]. Most of the CFD codes that were considered required initial- and boundary-condition arrays. Both specific and generic arrays were utilized. The specific arrays consisted of values appropriate to the computation, such as the boundary of a flow domain. Generic arrays were comprised of random numbers, linear counts, and/or constants. An intriguing empirical observation (based thus far on limited experience) is that the performance estimates generated by the simulator for these incomplete, often physically-meaningless input arrays, is that the actual performance, obtained from an exact simulation, is typically within 10% - 20% of the estimated performance.

3.1 Examples

To illustrate simple optimization and domain decomposition, consider the matrix multiply kernel:

```
c   4-way unrolled matrix multiply routine
      do 100 k = 1, n
        do 100 i = 1, l
          c(i,k) = 0.0
100     continue
      do 110 j = 1, m, 4
        do 110 k = 1, n
          do 110 i = 1, l
            c(i,k) = c(i,k) + a(i,j) * b(j,k)
     .               + a(i,j+1) * b(j+1,k)
     .               + a(i,j+2) * b(j+2,k)
     .               + a(i,j+3) * b(j+3,k)
110   continue
```

This code was input into the compiler with optimization disabled, to illustrate the basic treatment of inner DO-loop code. The corresponding WIC code produced by the compiler is

```
==0.0
==$3#4+$1#2*$2#3+$1#2*$2#3+$1#2*$2#3+$1#2*$2#3
```

which is then mapped on to the processors. Since it is natural for a programmer to think of data-flow-type architectures, such as the NSC, in a graphical sense, the compiler provides an optional graphical output of the pipelines that are formed in the resource allocation stage. As shown in Figure 1, the compiler outputs the ALS functionality and memory/processor interconnects ordered by the WIC:

When optimization is invoked, and the memory/processor model of the target architecture indicates that sufficient resources exist for parallel/vector processing, as is the case for the NSC, the compiler begins to implement various code transformations and non-standard heuristics to maximize the use of available assets, while maintaining an acceptable level[7] of memory conflicts. In the case of the matrix multiply example given above, the compiler effectively unrolls the loop, after testing a number[8] of code constructs and data mappings. (Note that this kind of loop-unrolling was not a heuristic presented to the compiler.) The compiler then generated the following code:

```
      do 100 k = 1, n
        do 100 i = 1, l
          c(i,k) = 0.0
100   continue
      do 110 j = 1, m, 4
        do 110 k = 1, n
          do 110 i = 1, l
            c(i,k) = c(i,k) + a(i,j) * b(j,k)
     .               + a(i,j+1) * b(j+1,k)
     .    + a(i,j+2) * b(j+2,k)
     .               + a(i,j+3) * b(j+3,k)
110   continue
                      o
                      o
                      o

      do 400 k4 = 1, n4
        do 400 i4 = 1, 14
          c4(i4,k4) = 0.0
400   continue
      do 410 j4 = 1, m4, 4
        do 410 k4 = 1, n4
          do 410 i4 = 1, 14
            c4(i4,k4) = c4(i4,k4)
                      + a4(i4,j4) * b4(j4,k4)
     .                + a4(i4,j4+1) * b4(j4+1,k4)
     .    + a4(i4,j4+2) * b4(j4+2,k4)
     .                + a4(i4,j4+3) * b4(j4+3,k4)
410   continue
```

As a representative example of PMPC use, the optimal design of an NSC node is considered. To

[5] Singlets, doublets and triplets refer to the number of floating-point units within a single processor group.

[6] Several 0:2:2/4 (i.e. two doublets, two triplets - 10 total floating-point units — and 4 memory planes) NSC Minimodes have been built at Princeton University

[7] As defined in a user-specified optimization-metrics list.

[8] Eight candidate 'optimizations' were attempted by the compiler in this example.

help determine an appropriate number and balance of memory and computational resources in a node, the PMP compiler was applied to various node configurations, with the number of memory planes and processors as primary parameters. An extensive investigation of memory-processor parameter space was performed. A suite of FORTRAN CFD test codes (including NAV3D, CRALE, and benchmarks such as the NAS kernels) were combined to yield over 10^4 source lines. 10^4 discrete *memory-processor* combinations and configurations were 'compiled' to the point of generating optimal code and data storage. The PMPC ran the raw *benchmark* code at a rate of 400 - 2,000 lines per minute on a VAXstation 3100. The result was the selection of a 4:8:4/16 processor node configu*ration* which had a projected sustained performance in excess of 70% peak. The compiler output was checked extensively for this particular configuration, since it is likely to be implemented in hardware.

4 Summary and Conclusions

It is surprising that the minimal simulator, coupled with small arbitrary input-data arrays, has apparently led to reasonable *good* performance estimates and statistics. Thus far the data leading to this observation are empirical, and based on a limited experience base. A sensitivity study of simulator is underway to address this result.

Although the PMPC is designed to support a number of parallel architectures, the main target to date has been a single NSC node. Given that the architecture of a node possesses both medium and fine granularity, and requires extensive internal data decomposition and and load-balancing among multiple (eg. 8-32) independent memory planes and multiple (eg. 16-64) processors, it may be argued that the PMPC has been demonstrated in a somewhat general, parallel environment. The PMPC is currently being expanded to accommodate multiple nodes with message-passing and direct-routing *internode* protocols. The main output of the compiler is a pseudocode from which detailed machine code may be, but thus far has not fully been, *derived* for direct execution on on hardware.

Acknowledgement

The author gratefully acknowledges the support of the Air Force Office of Scientific Research in providing Grant 91-0003 for the present work.

References

[1] W.S. Flannery, "Architecture and Applications of the Navier-Stokes Computer," *Ph.D. Thesis*, Department of Mechanical and Aerospace Engineering, Princeton University, 1991.

[2] M.E. Hayder, "The Navier-Stokes Computer," *Ph.D. Thesis*, Department of Mechanical and Aerospace Engineering, Princeton University, 1988.

[3] D.M. Nosenchuck, W.S. Flannery, M.E. Hayder, "A Navier-Stokes Computer," *Special-Purpose Computers*, B. Alder, ed., pp 97 - 134, 1988.

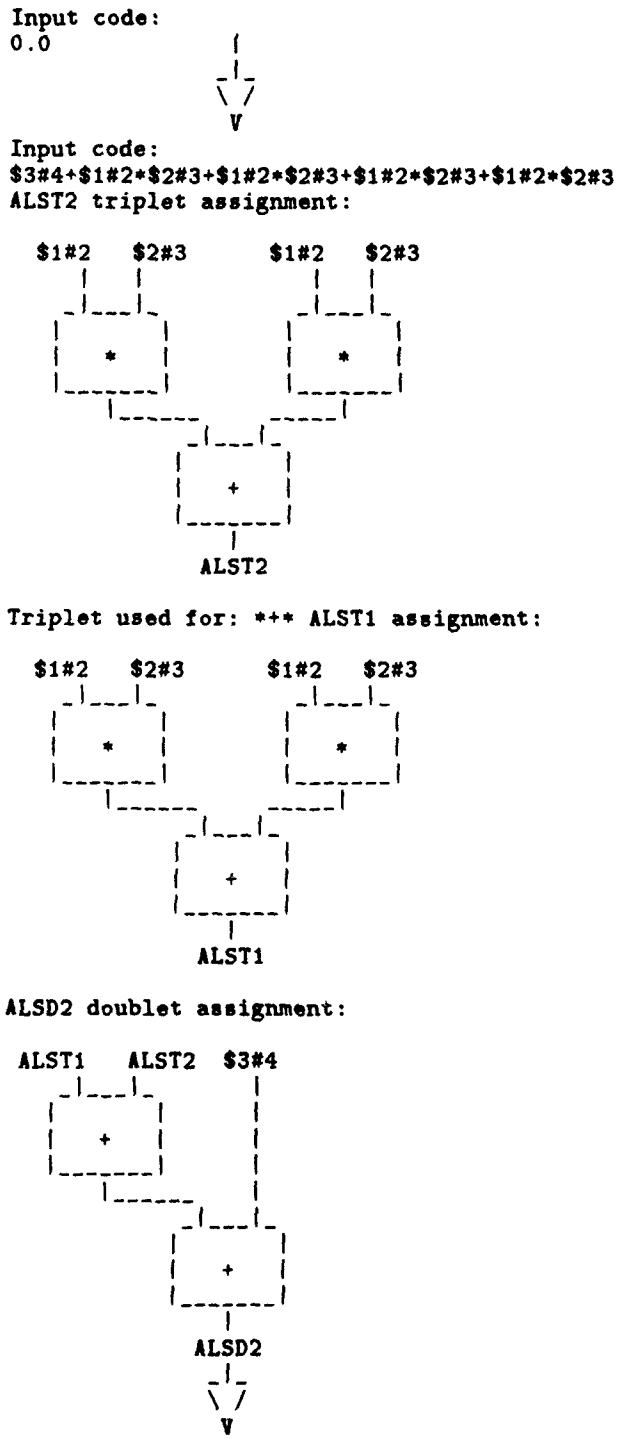

Figure 1: ALS Pipeline For MXM Baseline Subroutine